Prototyping Languages Related Constructs and Tools with Squeak*

Alexandre Bergel
Distributed Systems Group
Dept. of Computer Science
Trinity College Dublin 2, Ireland
www.cs.tcd.ie/Alexandre.Bergel

Marcus Denker
Software Composition Group
University of Bern
Bern, Switzerland
www.iam.unibe.ch/∼denker

May 23, 2006

Abstract

Prototyping new programming languages is often assimilated as a task requiring heavy expertise in parsing and compilation. This paper argues that choosing as a host platform a language having advanced reflective capabilities helps in reducing the effort and time spent on developing new language related constructs and tools.

The Squeak Smalltalk implementation provides very expressive reflective facilities. In this paper we focus on having methods as first class entities, enabling methods manipulation as plain standard objects and reification of method execution. Powerful language related tools and efficient new programming constructs can be quickly implemented. ByteSurgeon, a bytecode manipulation library, and FacetS, an aspect mechanism, serve as illustrations.

1 Introduction

Traditionally, developing a new programming language is a task that usually requires knowledge of complex techniques like lexical parsing and syntactic analysis, and compilation toward native or virtual machine code. Although these steps are commonly adopted when new languages are developed, this process is heavy and necessitates significant efforts which hamper experimentation.

Smalltalk has long offered powerful reflective capabilities [8] such as a metaclass hierarchy and first class entities for most of the elements of the language such as classes, method dictionaries and methods. Moreover, a powerful Smalltalk environment implementation offers a compiler, debugger and others tools that can be easily adapted. Smalltalk is therefore a natural choice to experiment and prototype new programming languages [1, 3, 4, 6].

Squeak [11, 15] is a Smalltalk implementation that brings reflection a step further. Whereas it handles *methods as first class entities* that can be modified and replaced on the fly, it also allows for *method execution reification*, exposing dynamic information related to method execution as objects.

This paper first describes how bytecode is stored as plain *compiled method* objects. This object defines a method and is associated with a name within a *method dictionary*. Each class is associated with a method dictionary. CompiledMethod and MethodDictionary are two classes present in most of the Smalltalk implementations (including Squeak), and allow for easy and powerful bytecode manipulation. This is illustrated by ByteSurgeon [7], a bytecode transformation framework.

Compiled methods contain bytecode and are directly interpretable by the Squeak virtual machine. However, if another object stands for a compiled method in a method dictionary, the virtual machine

*Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06), July 2006.

(VM) detects it and instead of directly interpreting it, the VM sends the message run: methodName with: listOfArguments in: receiver to this object. We call this mechanism *method execution reification*. This mechanism is used to implement FacetS, a dynamic aspect system with just three classes.

This paper claims that choosing a reflective language as a platform reduces effort and time spent in developing new syntactical language constructs and language related tools. The contributions of this paper are:

- Description of a runtime environment that supports methods as first class entity.
- Presentation of the *method execution reification* mechanism.
- Illustrations of this concepts with ByteSurgeon, a high level bytecode transformation framework, and FacetS, a minimal dynamic aspect system.

First, Section 2 describes the two reflective mechanisms of Squeak: methods as first class entities (Section 2.1) and reification of method execution (Section 2.2). Then in Section 3 the use of first class entity methods is illustrated with ByteSurgeon, and Section 4 illustrates with FacetS reification of method execution. And finally in Section 6 the conclusion is presented.

2 Reflective Capabilities of Squeak

Programming languages such as Smalltalk [9], CLOS [2], Ruby [19], Self [20] are often classified as dynamic languages referring to their dynamic type system, advanced reflective features, use of metaclasses and their interactivity with the programmer.

Squeak [11, 15] is an open-source Smalltalk dialect that offers expressive reflective features related to methods like reification of method and method execution.

2.1 Methods as Objects

Reification of Methods Classes encode the structure of instances: variables, superclass, subclass relationships and methods. In Squeak, both classes and methods are represented by first class objects that can be accessed and even changed at runtime.

Compiled methods are objects in Squeak that describe the executable part of methods. These contain a pointer to the sourcecode and the code executed by the virtual machine, the bytecode. These methods are stored in a dictionary (a *method dictionary*), keyed by their names. We can access the method dictionary of a class by sending the message methodDict to this class. For instance, executing ExampleClass methodDict return the methods dictionary of the class ExampleClass.

This dictionary then provides an interface for adding or replacing methods. When a new method is added from a programming tool (*e.g.,* a Smalltalk code browser) in essence the environment performs the following:

```
ExampleClass methodDict at: #myMethod
                         put: aCompiledMethod
```

A compiled method (aCompiledMethod) is added to the method dictionary of the class ExampleClass under the name myMethod. The effect is the addition of a new method.

2.2 Objects as Methods

Reification of Method Execution When a message named m is sent to an object, the corresponding method is searched along the single inheritance link of the class of the object. Once the method named m is found in one of the superclass (*i.e.,* its method dictionary contains an entry for m), then the corresponding value associated with m is fetched from the method dictionary by the VM.

Now the method gets invoked by the VM. One of two cases happens: either this method is a *compiled method* (*i.e.,* list of bytecode), or it is an *object method* (a plain object). If this value is a compiled method (an instance of the class CompiledMethod), then the VM directly interprets the bytecode. If this

value is *not* a compiled method, then the VM performs a *reification of the method execution*.

This method execution reification is performed by the VM when sending the message run: methodName with: listOfArguments in: receiver to this object method. In such a case, methodName contains the name of the method currently invoked, listOfArguments contains the list of arguments provided, and receiver a reference to the receiver to the message.

For instance, let assume two classes, C and Wrapper:

```
Object subclass: #C
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Example'.

Object subclass: #Wrapper
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Example'.
```

The class Wrapper contains a method named run:with:in: that takes three arguments, a name (methodName), an array of objects (listOfArguments), and an object reference (receiver):

```
Wrapper ≫run: methodName with: listOfArguments in: receiver
  "We first display some info on the standard output stream"
  Transcript show: 'Method ', methodName,
    ' arguments: ', listOfArguments printString,
    ' receiver: ', receiver printString.

  "Then we return the first element of the provided list"
  ^listOfArguments first
```

Prior to returning the first elements of listOfArguments, some information is written to the standard output stream.

For instance, a method foo: can be added to the class C by evaluating:

```
"We create an instance of Wrapper"
w := Wrapper new.

"We create an instance of C"
c := C new.

"We add a new method named foo: to C"
C methodDict at: #foo: put: w.
```

As similarly shown in Section 2.1, the last line above adds an entry named foo: to the method dictionary of C. Instance of C therefore understand messages named foo:. Because of the method execution reification mechanism, evaluating c foo: 10 is in fact evaluated by the VM as:

```
w run: #foo: with: #(10) in: c
```

The result is 10, and displays the following on the standard output stream:

```
Method foo: arguments: #(10) receiver: a C
```

3 Case Study: Bytecode Transformation

We have seen that replacing CompiledMethod objects is simple to do in Squeak. But sometimes, the granularity of change needed is not that of a method. We might want to modify a method by just adding or replacing some parts.

Examples for cases where this is useful are *e.g.,* adding hooks into bytecode for tracing and profiling. Bytecode transformation has a number of advantages over using other representations. Program text is not required and the performance of transformation is better: We are already working on the lowest level and thus skip the costly code generation phase of a full compiler.

Bytecode transformation is not limited to Squeak, it has been implemented before in the Java world [5]. But Squeak has one advantage: We can change methods at runtime, whereas Java is limited to do purely load-time transformations.

The following contains a short introduction of ByteSurgeon. More detailed description can be found in our previous work [7].

3.1 ByteSurgeon Overview

Before explaining how ByteSurgeon works, we will look at an example. Our goal is to edit bytecode inside a method by inserting new bytecodes before the send bytecode:

```
(Example>> #aMethod) instrument: [:instr |
    instr isSend ifTrue: [instr insertBefore: 'Counter inc']
].
```

The user of ByteSurgeon deals with methods: The method that should be edited is sent the instrument: message. This takes as an argument a block that is then executed for each bytecode in the method. ByteSurgeon takes care to do all decoding, it provides objects describing a bytecode, not raw numbers, as the parameter for the block.

With the bytecode objects the following operations can be performed: insertion before, after or replacement. The code to be inserted is described using a string of standard Smalltalk code.

The example thus will insert code for incrementing a counter in front of every virtual machine instruction *send* bytecode in the method #aMethod of the class ExampleClass.

ByteSurgeon provides other, more advanced features. The user can access certain values that turn out to be interesting, but hard to get to. For example, when replacing a VM instruction send bytecode, the receiver and the arguments of that send have been pushed on the stack. ByteSurgeon can provide access to these values, when needed, using so called *meta variables*. As a simple example, the following code provides a tracer with the receiver object of all sends:

```
(ExampleClass>> #aMethod) instrument: [:instr |
    instr isSend ifTrue: [
        instr insertBefore:
            'Tracer traceSendTo: <meta: #receiver> ']
].
```

ByteSurgeon takes care to generate code for saving the needed value for access in the inlined code.

3.2 Implementation

ByteSurgeon uses the backend of a Smalltalk source-to-bytecode compiler. This compiler provides an intermediate form called *intermediate representation* (IR) that abstracts from the specific bytecode details.

So we use the compiler backend to decompile the method object into the IR, then ByteSurgeon transforms the IR, and the codegenerator then generates a new compiled method (see figure 1).

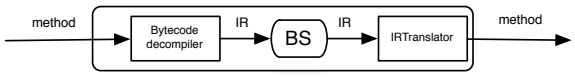

Figure 1: ByteSurgeon

For a more detailed description and benchmarking results see [7].

3.3 Projects using ByteSurgeon

ByteSurgeon has been used to implement Geppetto, a complete, fine-grained dynamic Meta Object Protocol [13] based on the design of Reflex [18]. As Reflex is based on Java, it can only provide reflective facilities when the bytecode has been transformed at loadtime. Gepetto on the other hand allows for completely unanticipated use.

Another application of ByteSurgeon is a trace-library for Squeak that is used in a number of contexts: Feature Analysis based on dynamic runtime traces and an ongoing project to implement an *Omniscient Debugger* [14] for Squeak.

4 Case Study: Dynamic Aspects with FacetS

Aspect-oriented programming (AOP) [12] holds the promise of composing softwares out of orthogonal concerns. AOP promotes code insertion (*i.e.,* advices) at some particular locations in the source code (*i.e.,* join point[1]). These advices are executed when the control flow reaches a join point. AOP is useful to separate non-functional cross cutting concerns from a base system like security, logging or persistence [12].

In this section, we describe the implementation of a dynamic aspect mechanism, called FacetS, made in Squeak. In order to keep the description short,

[1]For the sake of simplicity, we do stick to the terminology usually used in the AOP community by making no distinction between *join point* and *join point shadow*.

we restricted the number of supported join points to two: *before* and *after* a method invocation.

Before describing how FacetS is implemented, we first present a small example that counts the number of call of a method. To compute factorial, the Integer class has a method named factorial that does a self recursion. The following code defines a temporary variable nbOfCalls initialized with 0 and an advice composed of a piece of code (that increments the variable) and a pointcut which is a predicate identifying join point:

```
|nbOfCalls ad as |
nbOfCalls := 0.
ad := Advice new.
ad code: [:receiver :methodName :args |
        nbOfCalls := nbOfCalls + 1].
ad addPointcut: [:cls :methodName |
        (cls name = #Integer) & (methodName = #factorial)].
ad modifier: #before.
```

The code above means that the increment specified in the block will be executed before the method factorial defined in the class Integer. An aspect, composed of a set of advices is created by:

```
as := Aspect new.
as addAdvice: ad.
```

This aspect is then installed on the class Integer with: as installOn: Integer. Evaluating 10 factorial returns 3628800 and increment the nbOfCalls to 11. The aspect can be removed with: as remove.

Aspect. As we previously mentioned, an aspect is a set of advices that can be applied to a set of classes. The class Aspect has two instance variables, a method installOn: and a method remove. It is therefore defined as:

```
Object subclass: #Aspect
  instanceVariableNames: 'classes advices'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'FacetS'

Aspect >> installOn: aClassOrACollectionOfClasses
  |aCollectionOfClasses |
  aCollectionOfClasses :=
    aClassOrACollectionOfClasses isCollection
      ifTrue: [aClassOrACollectionOfClasses]
      ifFalse: [Array with: aClassOrACollectionOfClasses].
  self advices do: [:ad |ad installOn: aCollectionOfClasses]

Aspect >> remove
  self advices do: [:ad |ad remove]
```

Pointcut. A join point is an element of the language semantics that the aspect coordinates with. It traditionally identify location within the source code of an application. A *pointcut* is a predicate used to identity join points. In FacetS, a pointcut is represented as a function that takes as argument a class and a method, and return whether this particular method is a join point or not. In the example above, the pointcut [:cls :methodName |(cls name = #Integer) & (methodName = #factorial)] identity the method factorial on the class Integer.

Advice. An advice contains

- a list of pointcuts, *i.e.,* locations within the source code.
- some piece of code intended to be executed if the pointcuts are activated.
- a modifier (*i.e., before*, *after*, or *around*).
- the list of affected methods by this advice necessary for removal.

The definition of the class Advice is the following:

```
Object subclass: #Advice
  instanceVariableNames:
      'modifier pointcuts code methodsAffected'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'FacetS'
```

Installation of an advice is performed through the method installOn: taking a list of classes sent to an advice instance:

```
Advice >> installOn: aCollectionOfClasses
  methodsAffected := OrderedCollection new.
  aCollectionOfClasses do: [:c |
    c methodDict keys do: [:m |
      self pointcuts do: [:p |
        (p value: c value: m)
          ifTrue: [ |cm wrapper |
            cm := c methodDict at: m.
```

```
    methodsAffected add:
      (MethodReference class: c selector: m).
    wrapper := Wrapper
      advice: self compiledMethod: cm.
    c methodDict at: m put: wrapper]]]]
```

We first iterate over all the classes on which an aspect is installed. For each of these, we iterate overs its list of method names, and check if the pointcut is activated or not (*i.e.,* if the block that defines a pointcut is true when evaluated with the class and a method name). If it is the case, then we keep track of the affected method, and replace the method with a wrapper (discussed further in this section).

Removing a method is simply done by iterating over the affected methods and replacing the wrapper with its corresponding compiled method.

```
Advice >> remove
  methodsAffected do: [:mref |
    |c m w |
    c := mref actualClass.
    m := mref methodSymbol.
    w := c methodDict at: m.
    c methodDict at: m put: w compiledMethod]
```

Wrapper. A wrapper is an object intended to be inserted within a method dictionary in order for a method to be instrumented with an advice. A wrapper contains a reference to the advice from which it has been produced from and a reference to the compiled method that it instruments.

```
Object subclass: #Wrapper
  instanceVariableNames: 'advice compiledMethod'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'FacetS'
```

As shown in the installOn: method of the class Advice, when a method is instrumented, its compiled method is replaced by a wrapper that contains a reference to this compiled method and to an advice.

Sending a message for which a compiled method is not associated with triggers the *method execution reification* mechanism described in Section 2.2. When a method instrumented with a wrapper is invoked, the method run:with:in: defined on the wrapper is executed:

```
Wrapper >> run: aSymbol with: args in: rec
  |code modifier argsForCode |
  code := advice code.
  modifier := advice modifier.

  argsForCode := rec . aSymbol. args.

  (modifier isNil or: [modifier == #before])
    ifTrue: [code valueWithArguments: argsForCode.
      ^compiledMethod
          valueWithReceiver: rec arguments: argsForCode].
  (modifier isNil or: [modifier == #after])
    ifTrue: [|v |
      v := compiledMethod
          valueWithReceiver: rec arguments: argsForCode.
      code valueWithArguments: argsForCode.
      ^v].
```

Minimal AOP mechanism. This section fully describes a dynamic aspect mechanism consisting of 3 classes and 5 methods. Aspects can be installed and removed dynamically. The key point of this design is the method run:with:in: that execute some instrumentation before or after a method. This method is triggered by the VM, using the *method execution reification* mechanism.

5 Related Work

Section 2 presents two reflective mechanisms, however many others exist. This section gives a short description of the reification mechanisms provided by some other systems. For space reasons, we can not give a complete overview of related work.

Smalltalk. Few message passing control techniques other than the one presented in Section 2 are offered by Smalltalk [8] (and thus Squeak):

- By embedding a compiler in the Smalltalk environment, methods' source code can be easily modified. Methods can be recompiled on the fly.
- Error handling can be easily specialized. The idea is to specialize the doesNotUnderstand: method invoked when an object does not understand a message.
- Introduction of anonymous subclass into the inheritance hierarchy allows message defined in upper classes to be intercepted.

MetaclassTalk. An extension of Smalltalk named MetaclassTalk [3] provides explicit metaclasses and allows the method evaluation process to be transparently controlled and redefined. Within MetaclassTalk, classes play the role of meta objects which define execution mechanisms for their instance.

CLOS. Common Lisp has long offered an object model, CLOS. It is one of the few class based languages to offer the ability to define instance specific methods [2]. Moreover CLOS is also one of the rare languages to provide a meta object protocol in which message passing control is an entry point [13].

Java. The package java.lang.reflect has been distributed since the version 1.1 of Java. It provides few mechanisms to allows limited information about classes and object to be reified. However, it does not allow methods and classes to be modified at runtime. Reflex [16] is an extension of Java that offers a full meta object protocol using bytecode modification. As Java allows not to change methods at runtime, these modifications need to be done in advance at load time.

Reflex. In the context of Java, Reflex [17] provides building blocks for facilitating the implementation of different aspect-oriented languages so that it is easier to experiment with new AOP concepts and languages, and it is also possible to compose aspects written in different AOP languages. It is built around a flexible intermediate model, derived from reflection, of (point)cuts, links, and metaobjects, to be used as an intermediate target for the implementation of aspect-oriented languages.

AspectS. The first AOP system designed for Squeak is AspectS [10]. AspectS and FacetS differ regarding the aspect construction. With AspectS a new aspect is created by subclassing the class Aspect. An advice is then associated to a method.

With FacetS, an aspect is created by instantiating the class Aspect and by providing pointcuts and advices. FacetS supports incremental definition of aspects.

6 Conclusion

This paper presents and illustrates two reflective features of Squeak. First, methods are first class entities. As plain objects, methods can be replaced and modified on the fly. Tools such as ByteSurgeon, a high level bytecode transformation framework, are easily implemented. Most of Smalltalk systems support methods as first class entity. This is not a characteristic of Squeak, but we believe it is an important mechanism that is not widely known or accepted in the language community. The second reflective feature presented is *method execution reification*: the VM emits a plain message when a method that is not compiled into bytecode has to be executed. This message sent contains information regarding the current method execution. This feature is a characteristic of Squeak. This mechanism is used to develop FacetS, a minimal dynamic AOP systems consisting of 3 classes.

Each programming language has its own specificity, having therefore a limited domain of applicability. Prototyping new programming language allows for experimentation with new constructs where results have to be quickly produced. This paper argues that a language having extended reflective features constitutes a powerful host platform.

Acknowledgments. We would like to thank Andrew Jackson and Yong Wang for their valuable comments and Andreas Raab for designing and implementing Squeak's *method execution reification*.

We gratefully acknowledge the Science Foundation Ireland, Lero - the Irish Software Engineering Research Centre and the the financial support of the Swiss National Science Foundation for the project "A Unified Approach to Composition and Extensibility" (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006).

References

[1] A. Bergel, C. Dony, and S. Ducasse. Prototalk: an environment for teaching, understanding, designing and prototyping object-oriented languages. In *Pro-*

ceedings of ESUG International Smalltalk Conference 2004, pages 107–130, Sept. 2004.

[2] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common lisp object system specification, x3j13. Technical Report 88-003, (ANSI COMMON LISP), 1988.

[3] N. Bouraqadi. Concern oriented programming using reflection. In *Workshop on Advanced Separation of Concerns – OOSPLA 2000*, 2000.

[4] J.-P. Briot. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In S. Cook, editor, *Proceedings ECOOP '89*, pages 109–129, Nottingham, July 1989. Cambridge University Press.

[5] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *Proceedings of GPCE'03*, volume 2830 of *LNCS*, pages 364–376, 2003.

[6] P. Cointe. The ClassTalk system: A laboratory to study reflection in smalltalk. In *OOPSLA/ECOOP Workshop on Reflection and Metalevel Architecture*, 1990.

[7] M. Denker, S. Ducasse, and É. Tanter. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, July 2006.

[8] S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[9] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[10] R. Hirschfeld. AspectS – Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, number 2591 in LNCS, pages 216–232. Springer, 2003.

[11] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, Nov. 1997.

[12] G. Kiczales. Aspect-oriented programming: A position paper from the Xerox PARC aspect-oriented programming project. In M. Muehlhauser, editor, *Special Issues in Object-Oriented Programming*. Dpunkt Verlag, 1996.

[13] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[14] B. Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, Oct. 2003.

[15] Squeak home page. http://www.squeak.org/.

[16] É. Tanter, N. Bouraqadi, and J. Noyé. Reflex — towards an open reflective extension of Java. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 25–43. Springer-Verlag, 2001.

[17] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *LNCS*, Tallin, Estonia, sep 2005.

[18] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[19] D. Thomas and A. Hunt. *Programming Ruby*. Addison Wesley, 2001.

[20] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, Dec. 1987.